1. Tempo disponibile 120 minuti.
2. Non è possibile consultare appunti, slide, libri, persone, siti web, ecc.
3. Scrivere in modo leggibile, su ogni foglio, nome, cognome e numero di matricola.
4. Le soluzioni agli esercizi che richiedono di progettare un algoritmo devono:
   - spiegare a parole l'algoritmo (se utile, anche con l'aiuto di esempi o disegni),
   - fornire e commentare lo pseudo-codice (indicando il significato delle variabili),
   - calcolare la complessità (con tutti i passaggi matematici necessari),
   - se l'esercizio ammette più soluzioni, a soluzioni computazionalmente più efficienti e/o concettualmente più semplici sono assegnati punteggi maggiori.

1. Calcolare la complessità $T(n)$ del seguente algoritmo `mystery1`:

```
algoritmo mystery1(n: Int) --> Int
    int tot = 0, x = 0
    while (n > 1)
        n = n/2
        x = x+1
        tot = tot+mystery2(x)
    endwhile
    return tot

algoritmo mystery2(m: Int) --> Int
    int[1..m] A
    for i = 1..m
        A[i] = 0
    endfor
    return A[m]
```

Soluzione L'algoritmo `mystery1` utilizza `mystery2`. Iniziamo quindi l'analisi del costo computazionale $T'(m)$ associato alla chiamata `mystery2(m)`: tale algoritmo esegue operazioni di costo costante, ed include un ciclo che viene eseguito m volte, quindi $T'(m) = O(m)$. Consideriamo ora la chiamata `mystery1(n)`. Tale algoritmo include un ciclo che viene eseguito $\lfloor \log n \rfloor$ volte, e ad ogni ciclo viene invocato `mystery2(x)` con un valore x che inizialmente vale 1 e poi viene ogni volta incrementato di 1. Quindi il costo computazionale di tale chiamata risulta essere:

$$T(n) = \sum_{x=1}^{\lfloor \log n \rfloor} T'(x) = \sum_{x=1}^{\lfloor \log n \rfloor} O(x) = O(\frac{(\log n + 1)\log n}{2}) = O(\log^2 n)$$

2. Scrivere un algoritmo che prende in input un albero binario T e ne calcola il *diametro*, definito come il numero di nodi nel percorso più lungo che collega due foglie. Un albero con una sola foglia (o un solo nodo) ha diametro 1.

Soluzione Il diametro di un albero radicato in T si può calcolare come il valore massimo tra: (1) il diametro del sottoalbero sinistro di T, (2) il diametro del sottoalbero destro di T e (3) la somma tra l'altezza del sottoalbero sinistro + l'altezza del sottoalbero destro + 1. Quest'ultimo valore non deve essere tenuto in considerazione se il nodo T ha un solo figlio, in modo da gestire il caso in cui l'albero degenera in una lista.

Si può scrivere un algoritmo ricorsivo basato su una post-visita. Una soluzione efficiente calcola in un'unica 'passata' sia il diametro che l'altezza di un sottoalbero in modo da visitare i nodi una sola volta. Si noti che in questo caso l'altezza indica il numero di nodi che contribuiranno eventualmente al calcolo del diametro. L'algoritmo restituisce quindi una coppia di valori:

```
DIAMETRO-ALTEZZA(nodo T) --> [int, int]
   if (T == null)
      return [0,0];
   else {
      // calcola ricorsivamente diametro e altezza dei figli
      infoSX = DIAMETRO-ALTEZZA(T.leftChild);
      infoDX = DIAMETRO-ALTEZZA(T.rightChild);

      // per calcolare il diametro, se il nodo ha entrambi il figli o nessun figlio
      // si valuta anche il cammino che passa per il nodo stesso
      // altrimenti si valuta solo il diametro dell'unico figlio
      // (in questo caso uno tra infoSX[0] e infoDX[0] e' uguale a 0)
      if ( ((T.leftChild != NULL) & (T.rightChild != NULL)) ||
           ((T.leftChild == NULL) & (T.rightChild == NULL)) )
        diametro = max(infoSX[0], infoDX[0], infoSX[1] + infoDX[1] + 1);
      else
        diametro = max(infoSX[0], infoDX[0]);

      altezza = max(infoSX[1], infoDX[1]) + 1;

      return [diametro, altezza];
    }
```

La complessità risulta quindi essere O(N) visto che la visita attraversa i nodi una sola volta.

3. Un geologo effettua una escursione e deve raccogliere delle pietre mettendole in un contenitore. Il contenitore deve essere riempito esattamente per la sua capacità, identificata da un intero K. Durante il cammino incontra N possibili pietre; l'i-esima pietra incontrata ha peso identificato da un intero $A[i]$. Quando una pietra viene incontrata durante il cammino, questa può essere immediatamente inserita nel contenitore, oppure viene lasciata lungo il percorso. Esiste un vincolo: dopo aver inserito nel contenitore una pietra, tutte le pietre che verranno inserite successivamente devono avere un peso minore o uguale. Progettare un algoritmo che dati i valori K e N, ed il vettore $A[1..N]$, restituisce *true* se è possibile riempire il contenitore esattamente per la sua capacità, *false* altrimenti.

Soluzione Si può utilizzare la programmazione dinamica, considerando la seguente classe di problemi: $P(i,j)$=*true* se è possibile riempire un contenitore di capacità j usando come ultima pietra quella in posizione i, $P(i,j)$=*false* altrimenti. Una volta risolti tutti i problemi per $i \in 1 \dots N$ e $j \in 1 \dots K$, il problema iniziale coincide con il verificare se esiste $i \in 1 \dots N$ tale che $P(i,K)$=*true*.

Vediamo ora come risolvere questi sottoproblemi:

$$P(i,j) = \begin{cases} \textit{true} & \text{se } i = 1 \text{ e } j = A[1] \\ \textit{false} & \text{se } i = 1 \text{ e } j \neq A[1] \\ \textit{false} & \text{se } i > 1 \text{ e } j < A[i] \\ \textit{true} & \text{se } i > 1 \text{ e } j = A[i] \\ \exists l < i.\big(A[l] \geq A[i] \wedge P(l, j - A[i])\big) & \text{se } i > 1 \text{ e } j < A[i] \end{cases}$$

La definizione sopra deriva dal fatto che una soluzione al problema $P(i,j)$ prevede che $A[i] = j$ (si prende solo la i-esima pietra) oppure che esista una soluzione al problema $P(l, j - A[i])$ a cui si aggiunge la i-esima pietra (per poter aggiungere la i-esima pietra alla soluzione del problema $P(l, j - A[i])$, la i-esima pietra deve essere successiva rispetto alla l-esima, quindi $l < i$, e il peso della i-esima pietra deve essere minore o uguale al peso dalla l-esima, quindi $A[l] \geq A[i]$).

Cont.

Il seguente algoritmo risolve il problema dato, risolvendo prima tutti i sottoproblemi $P(i, j)$ utilizzando una struttura dati ausiliaria `M[1..N, 1..K]` per memorizzare le soluzioni ai sottoproblemi $P(i, j)$ (più precisamente, avremo `M[i,j]` $= P(i, j)$).

```
algoritmo contenitorePietre(K: Int, N: Int, A: Int[1..N]) --> Boolean
    Boolean M[1..N, 1..K]
    // inizializzazione prima riga di M
    for j = 1..K
        if (j == A[1])
            then M[1,j] = true
            else M[1,j] = false
        endif
    endfor
    // riempimento delle successive righe di M
    for i = 2..N
        for j = 1..K
            if (j < A[i]) then M[i,j] = false
            elseif (j == A[i]) then M[i,j] = true
            else
                Boolean flag = false
                for l = 1..(i-1)
                    if (A[l] >= A[i]) && (P(l,j-A[i])) then flag = true
                endfor
                M[i,j] = flag
            endif
        endfor
    endfor
    // verifica dell'esistenza di una soluzione (riempimento contenitore di capacita' K)
    for i = 2..N
        if M[i,K] then return true
    endfor
    return false
```

Valutiamo ora la complessità $T(K, N)$ di tale algoritmo. Il primo ciclo ha costo $O(K)$, il secondo è costituito da operazioni di costo costante eseguite all'interno di 3 cicli annidati (quello esterno viene eseguito $N - 1$ volte, il secondo K volte, ed il terzo viene eseguito al più N volte), mentre il terzo ha costo $O(N)$. Si può quindi concludere che $T(K, N) = O(K) + O(NKN) + O(N) = O(KN^2)$.

4. Un ordinamento topologico dei vertici di un grafo diretto aciclico $G = (V, E)$ è una visita dei vertici del grafo, tale per cui se esiste un arco $(u, v) \in E$ allora u viene visitato prima di v. Progettare un algoritmo che dato il grafo orientato G stampa i vertici del grafo secondo un ordinamento topologico.

Soluzione Si può procedere stampando prima i vertici senza archi entranti, e poi per ogni vertice stampato si eliminano gli archi uscenti da esso, procedendo a stampare anche gli altri vertici nel momento in cui vengono eliminati tutti i suoi archi entranti. Da un punto di vista algoritmico, si può inizialmente calcolare per ogni vertice il grado in ingresso, ovvero il numero di archi entranti. Si memorizzano in una coda i vertici con grado in ingresso uguale a zero (essendo il grafo aciclico esiste sicuramente almeno un nodo con tale proprietà). Poi si iniziano ad estrarre vertici dalla coda; ad ogni estrazione si stampa il vertice estratto, e si decrementa a tutti i vertici adiacenti il grado in ingresso. Quando il grado in ingresso di un vertice diventa 0, tale vertice viene inserito in coda.

Il sequente algoritmo assume che i vertici del grafo $G(V, E)$ siano identificati dagli interi nell'intervallo $1 \ldots |V|$. Inoltre, si utilizzano `Q` come coda, e `inDegree[1..|V|]` come vettore per memorizzare i gradi in ingresso dei singoli vertici.

```
algoritmo ordTopologico(G(V,E))
    Queue Q
    Int inDegree[1..|V|]
```

Cont.

```
// inizializzazione inDegree
for v = 1..|V|
    inDegree[v] = 0
endfor

// calcolo dei gradi in ingresso
for each (u,v) in E
    inDegree[v]++
endfor

// inserimento in coda dei vertici iniziali
for v = 1..|V|
    if (inDegree[v] == 0) then Q.enqueue(v)
endfor

// visita topologica
while (! Q.isEmpty())
    Vertex v = Q.dequeue()
    print v
    for each w in v.adjacent
        inDegree[w]--
        if (inDegree[w] == 0) then Q.enqueue(w)
    endfor
endwhile
```

Valutiamo ora la complessità $T(n, m)$ di tale algoritmo, assumendo $n = |V|$ e $m = |E|$. Il primo ciclo ha costo $O(n)$, il secondo $O(m)$, il terzo $O(n)$ ed il quarto $O(n+m)$ in quanto considera tutti i vertici e tutti gli archi una ed una sola volta. Possiamo quindi concludere $T(n, m) = O(n + m)$.

The End.